Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis

Emil Dumitrescu, Alain Girault, Eric Rutten

Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis *

Emil DUMITRESCU, Alain GIRAULT and Eric RUTTEN
INRIA, Montbonnot, FRANCE

Abstract

Discrete controller synthesis is used to formally assess the fault-tolerance capabilities of a dependable system from the early design stages. Fault-tolerance is the ability of the system to handle failures so that its service and/or performance is maintained. Our originality is to use discrete controller synthesis to act on an executable specification in order to yield a new fault-tolerant executable specification, possibly non-deterministic. Then, we obtain manually the final distributed implementation and we formally verify it against the initial fault-tolerant specification.

1 Introduction

This work addresses the fault-tolerance issue in discrete-event synchronous systems design. Fault occurrence must be taken into account from the early design stages of dependable complex systems. First, the designer must think about the functionalities of the system that may fail. Then, for each possible failure, a tolerance policy must be specified in order to limit error propagation (and prevent the failure of other components) and to ensure the correct behavior of the system, despite the failure. At the functional level, a fault-tolerance policy is described as an additional behavior that should be enabled whenever a failure occurs. Its purpose is the total or partial compensation of the loss of functionality/performance triggered by the failure.

However, given the complexity of an executable description, programming a fault-tolerance policy is a very difficult task. Indeed, the designer must think of all possible evolutions that may follow the failure of a component, and distinguish which ones must be avoided. Such an approach is not realistic, as it amounts to finding all possible execution sequences that satisfy a correctness requirement of a design. In this context, we propose a formal analysis framework, allowing designers to deal with fault tolerance at the functional level of the design executable specification. The main interest of working with executable specifications is that formal techniques are much more efficient at this level, as specifications are usually simple and non-deterministic.

We propose to use a particular *discrete controller synthesis* technique [7] to generate automatically, if it exists, a behavior implementing the user-specified fault-tolerant policy, and to compose it with the design's executable specification, in order to yield a new *fault-tolerant specification*. If such a behavior does not exist, this means that, at its current description level, the design is not fault-tolerant: there is no means to prevent failure propagations or inconsistent executions. The executable specification should hence be redesigned.

We claim that most fault-tolerant behaviors can be interpreted in terms of *total or partial service continuity*, according to the initial formal specification of the design. Ideally, we would like to formally state *what the nominal service of a design is*, and require that this nominal service be maintained for any expected fault occurrence. Discrete controller synthesis should then ensure the continuity of this service.

The resulting fault-tolerant specification gathers a set of possible fault tolerant behaviors. The choice between these behaviors is often non-deterministic. In order to derive a deterministic fault-tolerant implementation, a definitive choice must be made among the existing behaviors. Moreover, the implemented behavior must comply to the architectural constraints of the implementation. In practice, distributed control solutions are often required, because the target architecture is distributed. In order to handle distribution, we chose to address the implementation step *manually*. Indeed, the automatic generation of *communicating* distributed controllers is the only practical alternative to the manual distribution, and we show that it would not be a good choice given our fault-tolerance goal, which prohibits the introduction

*This work was supported by the *ARTIST* European project (IST-2001-34820)

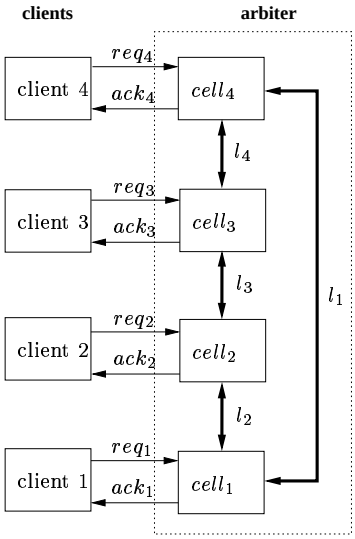

Figure 1: Architecture of the distributed arbiter

of additional communication.

Once an implementation has been manually obtained, its fault-tolerance properties can be assessed by comparing it formally with respect to the fault-tolerant specification it was derived from. We propose to use the *refinement checking* technique [2] in order to achieve this last step.

In the sequel, we present our general design approach and illustrate its different steps on a running example: the functional specification of a fault-tolerant distributed arbiter. Then, we discuss the results of our method as well as the manual implementation aspects, and we conclude on the advantages of our approach.

2 Specification of the distributed arbiter

The arbiter model we use as a running example is inspired from [8]. It deals with four clients (Figure 1) wishing to access some shared resource. Each client i ($i = 1..4$) can request access by asserting the input signal req_i. The arbiter replies accordingly by asserting the proper acknowledgment signal ack_i and possibly de-asserting other acknowledge signals $ack_j, j \neq i$. The arbiter must achieve two basic functionalities, mutual exclusion and liveness, formally expressed with the classical temporal logic CTL [1]:

- **Mutual Exclusion (ME)**: at any moment, at most one acknowledge signal can be asserted: $ME : AG(not(ack_1.ack_2 + ack_1.ack_3 + ack_1.ack_4 + ack_2.ack_3 + ack_2.ack_4 + ack_3.ack_4))$
- **Liveness (LIV)**: for each client i, if a request is asserted, then it is eventually acknowledged: $LIV_i : AG(req_i \rightarrow AF(ack_i))$

The arbiter does not memorize incoming requests. Thus, to satisfy LIV_i, we must assume that each client is persistent: it maintains req_i at least until receiving ack_i. However, if req_i is deasserted before obtaining ack_i, then it shall not be acknowledged at all. In order to implement liveness, a token mechanism is used. Each cell is periodically given priority with respect to the others. Each token is valid for only one single time unit (execution cycle) and then expires even though it has not been consumed. At any time, at most one token may circulate through the cells. The token is transmitted via the point-to-point dedicated communication links l_i ($i = 1..4$) established between consecutive cells. The cell i may only acknowledge its client if it is holding the token at the current cycle. However, in case $cell_i$ is not currently holding the token, then it may still acknowledge its client according to a weak priority policy: the cells are ordered according to a static priority scheme, from $cell_1$, which has the highest priority to $cell_4$. If the cell currently holding the token does not have an incoming request, then other cells must be able to acknowledge their incoming requests, provided mutual exclusion is respected. Thus, if $cell_i$ receives an incoming request while not holding the token, then it may acknowledge this request only if there are no more prioritary requests (cells $j = 1..i-1$) and if none among the less prioritary cells ($k = i+1..4$) is about to acknowledge its corresponding request while holding the token.

Finally, we wish our arbiter to have a distributed architecture: four cells, each corresponding to one client.

3 Overview of our method

3.1 Faults, errors, and failures

This is the fundamental causal principle in the study of fault pathology and the design of dependable systems [4]. A fault may yield an error: a physical hazardous event may yield a logical internal inconsistency of the system. An error may yield a failure of one or more components. A failure of one component can be viewed as a fault at the system level. Each possible failure is permanent.

In our context, we treat primary faults that can only be physical external events. We limit our study to *error processing*: we assume there exists a reliable external unit, which reports an error if and only if a physical fault has occurred. Failure management may involve both hardware and software or temporal redundancy. In our context, we rely on the manual introduction of hardware redundancy, followed by the automatic generation of redundant software, which exploits both the existing and the redundant hardware.

The design of dependable systems calls for a dedicated specification and validation procedure. Three

key points must be taken into account: the failure hypothesis, the failure model and the fault tolerance policy. In this section, we outline these three points from a general point of view, before specifying them formally with respect to our particular problem in the next section.

3.2 Defining a *failure hypothesis*

A failure hypothesis states which components of the system may fail. If more than one component is likely to fail, *failure configurations* are a common way to express subsets of components that may fail together. According to this hypothesis, the remaining components are supposed to be reliable: they never fail, or if such a failure occurs, the whole system fails. The failure hypothesis needs to be confirmed by a stochastic analysis step, in order to find the probability for each failure configuration. In this document, we assume that all failure configurations specified are equally probable.

3.3 Defining a *failure model*

When a component fails, specify what this failure implies. This amounts to defining a behavior that is triggered by this failure and that might influence the behavior of the remaining components. For instance, when a component fails (processor, communication link, sensor, etc.), it may stop reacting to its environment. This means either sending a constant value, or sending random values to the environment (Byzantine behavior), or sending no value at all (fail-safe behavior).

3.4 Defining a *fault tolerance policy*

Ideally, a fault-tolerant system should maintain its functionalities and its performance (nominal service) even though some of its components are down. In practice, this assumption is too strict and expensive to implement. Thus, in case a failure occurs, the nominal service may be replaced by a *degraded operating mode*. When the system runs inside a degraded mode, only a subset of its initial functional requirements are still met. We wish to achieve fault-tolerance by settling such a degraded operating mode when a failure occurs. In our context, this is done by expressing the appropriate constraints that are used to control the system's behavior from outside, in order to avoid unwanted configurations that may become reachable in case of a failure. This approach only applies to systems that are *controllable*: for such systems, a set of dedicated *controllable inputs* is provided for service maintenance purposes. The system behavior is constrained by driving the controllable inputs appropriately. All remaining input variables are called *uncontrollable*. The discrete controller synthesis technique [7] is an excellent choice for automatically producing such controlling constraints.

In our context, adding controllability to a design amounts to adding supplementary hardware (input variables and additional states and transitions) for backup purposes. This is synonym of specifying a hardware redundancy mechanism. On the other hand, the resulting synthesized controller, if it exists, represents a "program" (software redundancy) that appropriately uses the *already existing* redundant hardware in order to establish the degraded operating mode.

Discrete controller synthesis needs a formal *control objective*. In our framework, this control objective must express fault tolerance. This can be achieved in two ways:

1. Specify how a failure should be handled. This can be done whenever the back-up procedure is known a-priori, or when this procedure must comply to a pre-defined standard. The resulting controller should drive the controllable inputs accordingly.
2. Specify what the system should always do, despite failure occurrences. A degraded service is formally specified and the resulting controller must constrain the system so that this service is achieved.

4 Formally specifying the arbiter dependability

4.1 Failure hypothesis

We assume that only the communication links l_i between cells may fail. When link l_i fails, it stops transmitting tokens to the cell i. Each link is fail-silent and its failure is permanent. Thus, potentially, the cell i may never be acknowledged anymore. Moreover, we shall assume that the probability to have a failure affecting more than one link is negligible.

4.2 Link failure model

An abstract model for each communication link is given Figure 2. Each link may be either functional, in which case it transmits correctly the tokens and priority management information, or in a failure state, in which case it stops transmitting forever. The transition to the failure state is triggered by the environment and corresponds to a physical damage event.

The static priority between the cells is defined using an override/grant mechanism. Thus, cell i overrides cell $i+1$ iff req_i is active or if cell $i-1$ overrides cell i. The first cell in the chain (cell 1) cannot be overridden. On the other hand, cell i grants cell $i-1$ iff it is granted itself by cell $i+1$ and if it is not holding both an incoming request and a token. The last cell in the chain (cell 4) is always granted.

Each cell i is aware of the state (running or failure) of the communication link l_i it is connected to.

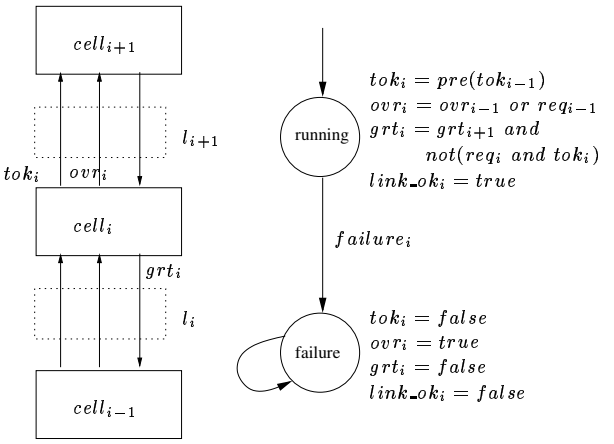

Figure 2: Abstract model of a communication link l_i

When link l_i is down, cell i stops receiving tokens. Thus, it stops complying with the specification LIV_i. Moreover, a link failure invalidates the static priority scheme, as no cell can be aware anymore of other requests of higher/lower priority.

4.3 Fault tolerance policy

When link l_i is down, the token keeps moving until it reaches l_i, and then it is lost. From that moment, no other request will ever be acknowledged. Thus, the mutual exclusion requirement is still trivially satisfied, but the response time is no longer guaranteed for any cell. Also, the static priority scheme is disabled.

The fault-tolerance policy must address the response time problem, while still ensuring mutual exclusion. As we know that in its degraded mode, our arbiter does not comply with the response time requirement, we wish to appropriately constrain its behavior, so that the response time becomes acceptable. We shall use the controller synthesis technique in order to produce appropriate controlling constraints. However, in its current design state, our arbiter has very few possible behaviors and can hardly be controlled: it should be avoided to constrain the incoming request or error signals, as requests and error events should be able to arrive at any moment. Besides, the only way to constrain the behavior of a reactive system is via its input signals. Thus, in order to achieve this fault-tolerance policy, we must redesign our arbiter interface and behavior, to make it more "controllable".

Adding controllability. When a link is down, the arbiter stops complying to its specification as soon as the token is lost. Thus, a possible supplementary control solution involves adding a *backup token* insertion point mechanism (Figure 3). The arbiter environment can reinsert a new token by adequately controlling this supplementary input. However, if l_i is operational, backup token reinsertion has no effect.

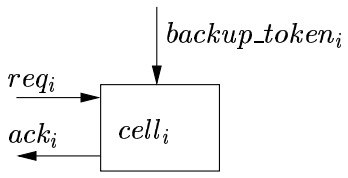

Figure 3: Cell i interface with hardware redundancy

We redesign the arbiter by assuming the existence of an external global *supervisor*. By asserting the $backup\_token_i$ signal, this supervisor inserts a new token inside the arbiter. By deasserting the $backup\_token_i$ signal, it prevents a new token from being inserted. First, backup tokens should only be inserted when at least one link is down. Second, when this is the case, tokens should not be inserted too often, as at most one token should circulate inside the arbiter. Finally, token insertion should be done periodically, so that each incoming request is acknowledged with a reasonable response time.

4.4 Achieving liveness control objectives

Up to now, discrete controller synthesis handles safety and non-blocking objectives. In order to address the liveness enforcement problem, we have chosen to replace the concept of "response in finite time" by a *bounded time* response requirement, which is more easily solvable by currently available controller synthesis algorithms. This requires finding a reasonable bound on the response time for each ack_i signal. Since there are four cells, and since at each cycle a new cell receives the token, we set the response bound to four cycles. Thus, the liveness requirement adapted to bounded time becomes:

$$LIV_i' : AG(req_i \rightarrow ack_i + AX(ack_i + AX(ack_i + AX(ack_i + AX(ack_i)))))$$

By combining the objectives ME and LIV_i' ($i = 1..4$), we obtain a controller having the following features:

- it observes every variable in the system;
- it is only allowed to control the inputs $backup\_token_i$ ($i = 1..4$) in order to constrain the arbiter to achieve its functional requirement;
- according to the control objectives, if link l_i fails, the controller will reestablish the global system coherence by possibly reinserting backup tokens through cell i, while preventing such token insertions whenever this would violate the ME requirement.

5 Validation of the fault tolerant behavior

We have modeled our arbiter and we have validated its fault-tolerance capability, by actually synthesizing a controller which ensures all the requirements stated in Section 4. For example, if link l_1 fails, the following scenarios are possible:

- req_1 is asserted and req_2, req_3, and req_4 are all deasserted. req_1 is acknowledged.
- req_1 is asserted. If req_2, req_3, and req_4 are asserted and if links l_2, l_3, and l_4 are operational, then the controller enforces the fair acknowledgment of all the requests, by inserting a backup token through cell 1. The token insertion takes into account the response delay bound of each cell. Thus, a possible future violation of the delay response bound is *anticipated* and avoided, by enforcing the insertion of a token at the appropriate moment.
- If cell j ($j = 2..4$) holds a token, then the controller disables the insertion of a new backup token until the currently existing token is lost inside the failed link l_i.

The controlled arbiter obtained is *non-deterministic*. The control which has been synthesized is called *maximally permissive*: controllable inputs are only constrained in case of necessity with respect to the control objectives. Indeed, if link l_i is operational, the value of the controllable input $backup\_token_i$ does not influence the behavior of the arbiter, as we chose by construction. If link l_i is down, the input $backup\_token_i$ only needs to be constrained in two specific scenarios:

- As long as the token ring contains a token, $backup\_token_i$ is set to $false$.
- When the token ring becomes empty (the token gets lost by link l_i), the controller enforces the introduction of a new backup token only when a possibility exists that a request is not acknowledged within the 4-cycle time bound. If only one incoming request is active at a time, it is immediately acknowledged, without a need for a token. Still, in such a situation, a token may be inserted.

The resulting controlled arbiter actually offers a choice among several available fault-tolerant behaviors. Because of its non-deterministic nature, we consider this result as a newly generated *executable specification* which embeds a set of fault-tolerant behaviors.

6 Implementing and verifying the fault-tolerant arbiter

At this stage, obtaining a possible implementation of the fault-tolerant arbiter would involve choosing *one* interesting behavior among those allowed by the executable specification and expressed in the controlled arbiter, with the additional constraint that the result must be *deterministic*.

The architecture of the resulting controller must also comply to the initial failure hypothesis. According to this hypothesis, only communication links may fail. Thus, we must assume that the controller cannot fail. Unfortunately, the resulting controller achieves *global control and observation*; thus, such an assumption is not realistic with respect to the distributed architecture of the arbiter. Hence, the implementation of the controlled arbiter must also meet the architectural constraint: *each cell i should be locally controlled*, and the only variables the local controller is allowed to monitor should be the internal variables of cell i. The automatic generation of local controllers achieving global control objectives is a much more difficult task, known as *decentralized controller synthesis* [5]. The existence of a distributed control solution often requires supplementary communication between the local controllers. In our context, assuming *reliable* communication between local controllers would contradict our initial failure hypothesis, which states that communication between cells is not reliable. On the other hand, the distributed controller synthesis problem without communication between local controllers has been shown to be undecidable [9].

As a consequence, we build the local controllers manually. By construction, our arbiter has several interesting properties. First of all, the controllable inputs are "don't cares" as long as no link failure has occurred. Then, according to the failure hypothesis, only one link may fail. Thus, when link l_i is down, only its corresponding $backup\_token_i$ needs to be controlled. Finally, by running a sufficient set of simulation scenarios, we realize that there exists a set of traces that are allowed by the controlled arbiter specification and that can be played on $backup\_token_i$. All these traces are composed by the following successive parts: (1) an arbitrary length prefix, corresponding to the normal operation, before any link failure occurs; all along this prefix, $backup\_token_i$ can be set to any value; (2) once a failure occurs, a bounded-length prefix follows, corresponding to the token propagation until it is lost; (3) finally, a periodic suffix corresponding to the periodic reinsertion of a new token inside the arbiter once the previous one is lost.

Such a behavior is *local* to each cell and can be reproduced by a simple manually built deterministic finite state machine, such as the one presented Figure 4. Hence, by composing the arbiter with four instances of this state machine, fault-tolerance is achieved, provided that only a single fault may oc-

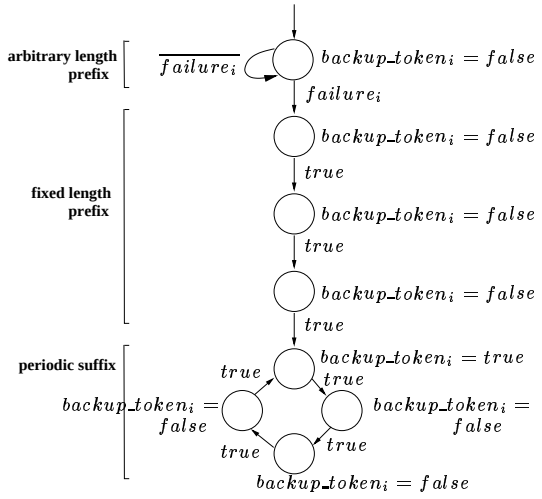

Figure 4: Possible implementation of a local controller for cell i

cur.

Assessing the correctness of the above implementation is a straightforward task and can be done by simulation. However, in general, such an exhaustive verification task can become very difficult as soon as the design becomes more complex. Formal verification techniques are known to deal more efficiently with this problem, even when applied on real-life designs. In particular, the *refinement checking* technique [2, 3] is used for assessing implementations against a non-deterministic operational specification. The use of this technique in our context is effective for two main reasons: first of all, if an implementation *refines* a specification, then it inherits every $\forall-$CTL property satisfied by the specification [2]. Note, however, that this conservative feature only concerns properties written in $\forall-$CTL. In our context, the fault-tolerance objectives ME and LIV_i, ($i = 1..4$) are compliant to the $\forall-$CTL subset of CTL. Second, refinement checking is a mature procedure which can extensively rely on compositionality, and whose complexity does not exceed the reachable state space computation of the implementation [3].

7 Discussion and conclusion

The main modeling benefit of our technique comes from the ability to specify what the nominal service to be maintained is, despite fault occurrences, and to automatically generate the behavior that maintains this service. The use of the discrete controller synthesis technique is very important in our framework. Even for a simple design like the distributed arbiter, finding a global fault-tolerance policy is a tricky task, and needed several synthesis attempts. Indeed, in the preliminary attempts an absence of solution was reported, which was mainly related to the design choices made concerning control freedom. This game solving approach is vital in our context, as it is the only way of determining whether a winning strategy exists for achieving fault-tolerance. This issue cannot be addressed using traditional model checking tools, as they do not distinguish between controllable and uncontrollable input events. To the best of our knowledge, the only available model checking tool that can also solve this question is MOCHA [3]. However, this model checking tool only checks the existence of a solution, without constructing it exhaustively, while the discrete controller synthesis based method is constructive.

Discrete controller synthesis enabled us to validate a functional executable specification by providing an exhaustive set of realistic simulation scenarios, which would have been difficult to set up manually. We have used this technique as a design tool for assessing the fault tolerant capabilities of a given executable specification. At the specification level, this assessment is only behavioral: "given its current design state, is my specification able to deal with faults, and if yes, how?". The synthesized controller answers this question.

The whole experimentation part has been realized using the SIGALI [7] discrete controller synthesis tool, as well as the mode automata under the MATOU [6] environment.

References

[1] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, number 131 in LNCS, New-York, 1981.

[2] O. Grumberg and D.E. Long. Model checking and modular verification. In *Proceedings of CONCUR'91*, volume 527 of *LNCS*, 1991.

[3] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of CAV'98*, pages 440–451, 1998.

[4] P. Jalote. *Fault-Tolerance in Distributed Systems.* Prentice Hall, 1994.

[5] F. Lin and W. M. Wonham. Decentralized supervisory control of discrete-event systems. *Information Sciences*, 1988.

[6] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou: An implementation of mode-automata into DC. In *Compiler Construction.* Springer verlag, 2000.

[7] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[8] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[9] S. Tripakis. Decentralized control of discrete event systems with bounded or unbounded delay communication. In *Proceedings of WODES'02*, Zaragoza, Spain, 2002.